Using Java Reflection to Automate Extension Language Parsing

Dale Parson (dparson@lucent.com)

Bell Laboratories, Lucent Technologies

Abstract

An extension language is an interpreted programming language designed to be embedded in a domain-specific framework. The addition of domain-specific primitive operations to an embedded extension language transforms that vanilla extension language into a domain-specific language. The LUxWORKS processor simulator and debugger from Lucent uses Tcl as its extension language. After an overview of extension language embedding and LUxWORKS experience, this paper looks at using Java reflection and related mechanisms to solve three limitations in extension language - domain framework interaction. The three limitations are gradual accumulation of ad hoc interface code connecting an extension language to a domain framework, over-coupling of a domain framework to a specific extension language, and inefficient command interpretation.

Java reflection consists of a set of programming interfaces through which a software module in a Java system can discover the structure of classes, methods and their associations in the system. Java reflection and a naming convention for primitive domain operations eliminate ad hoc interface code by supporting recursive inspection of a domain command interface and translation of extension language objects into domain objects. Java reflection, name-based dynamic class loading, and a language-neutral extension language abstraction eliminate language over-coupling by transforming the specific extension language into a run-time parameter. Java reflection and command objects eliminate inefficiency by bypassing the extension language interpreter for stereotyped commands. Overall, Java reflection helps to eliminate these limitations by supporting reorganization and elimination of hand-written code, and by streamlining interpretation.

1. Introduction

This paper examines the design of a set of Java utility interfaces and classes that simplify the work of integrating an extension language into a Java-based domain framework. Section 2 examines the role that extension languages play in extending domain-specific application frameworks. Section 3 looks at a commercial domain framework - extension language from Lucent called LUxWORKS. Experience in designing, building and maintaining LUxWORKS has led to this current research project within Bell Labs. Section 3 examines some limitations that have surfaced in the original, C++-based implementation of LUxWORKS. Section 4 follows through by eliminating these limitations, using a combination of Java reflection, Java dynamic class loading, a language-neutral extension language abstraction, and a set of naming conventions. Section 5 looks at related extension language-Java efforts. Section 5 also summarizes.

2. Extension Languages for domain-specific software systems

An *extension language* is a programming language that extends a domain-specific software application, tool or framework (hereafter "framework"). Interpreted languages such as Scheme [1], Tcl [2], or Python [3] often serve as extension languages because their interpreters support interactive creation and execution of custom extensions by framework users at run time. Many proprietary, framework-specific extension languages have come and gone, but with the maturation of extension language technology and the mass acceptance of so-called *scripting languages* [4], there is now seldom a need to invent a new extension language for an interactive framework. The term *command language* highlights the fact that an extension language usually adds imperative commands to a framework.

An extension language provides an application programming interface (API) that supports connections between the extension language and the *system programming language* [4] such as C, C++ or Java that implements the basic capabilities of the domain-specific framework. An extension language supports three categories of extensions:

- A framework extends an extension language by adding *domain-specific primitives* to the extension language's instruction set.
- Conversely, an extension language extends a domain-specific framework by adding an interpreted language capability.
- An extension language user extends the composite framework-language by writing extension functions in the extension language.

A framework engineer codes domain-specific primitives in a system programming language for efficiency, for security, and for compatibility with existing code libraries. Extension language designers intend their languages to be extended with new primitives; extension languages differ in this way from languages whose definitions are frozen by standardization. A primitive becomes an integral part of the extension language. If the extension language supports *dynamic loading* of primitives, then even users in the field can extend a framework-language system. Otherwise framework developers must add primitives via static linking.

Figure 1 illustrates the major calling relationships in a framework-language system. A user interface or extension language program (a.k.a. "script") passes textual extension language expressions to the extension language's *eval* primitive, named after the classic LISP eval that evaluates a textual expression [5]. Eval tokenizes and parses the expression. Eval then executes the functional pieces of the expression by calling *apply* (again from the LISP legacy) with a function and its arguments as parameters. Apply calls a primitive function directly. Apply invokes a function written in the extension language by binding formal parameters to arguments and calling eval recursively with the text of the interpreted function. In extension languages that support incremental compilation of intermediate code (a.k.a. *byte code*), apply invokes a function written in the extension language by calling an intermediate code interpreter with the intermediate code of the compiled function.

In a purely functional, LISP-like language, the result returned from eval is the result of the outermost function invocation in the expression. Non-functional languages may include operators that do not reduce to primitive function calls; eval interprets these directly.

Eval builds atop generic extension language primitives as well as domain-specific primitives. Primitives build atop library classes and functions in their respective modules. In addition, domain code can call extension language primitives directly, without the overhead of eval-based parsing. Domain code can also call extension language library code, for example string or hash table utility functions. Finally, eval itself is a primitive, allowing nesting of expressions within expressions and within domain data structures to arbitrary depth. One typical use of nesting is attachment of an extension language *callback expression* to some condition in the domain framework. When the domain meets that condition, it triggers a *callback event* to the extension language, and the extension language evaluates the expression as part of domain execution. Event-directed control is found in graphical user interfaces, in simulation systems, in loosely coupled distributed computing, and in the JavaBeans programming environment [6].

3. Tcl and the LUxWORKS embedded system simulator and debugger

3.1 Tcl-Luxdbg architecture

The research project of this paper grew out of experience in the architecture, design, implementation and maintenance of Lucent's *luxdbg* simulator and debugger for embedded processors [7]. Luxdbg uses Tcl as its extension language [8]. For an overview of luxdbg's architecture and design patterns see [9].

Figure 2 shows Tcl applied to luxdbg. Luxdbg registers the names and C++ function addresses of its primitives with Tcl at initialization time. Command strings drive Tcl. Tcl performs its language-specific manipulations on command strings, including variable substitution and concatenation of strings returned from nested Tcl

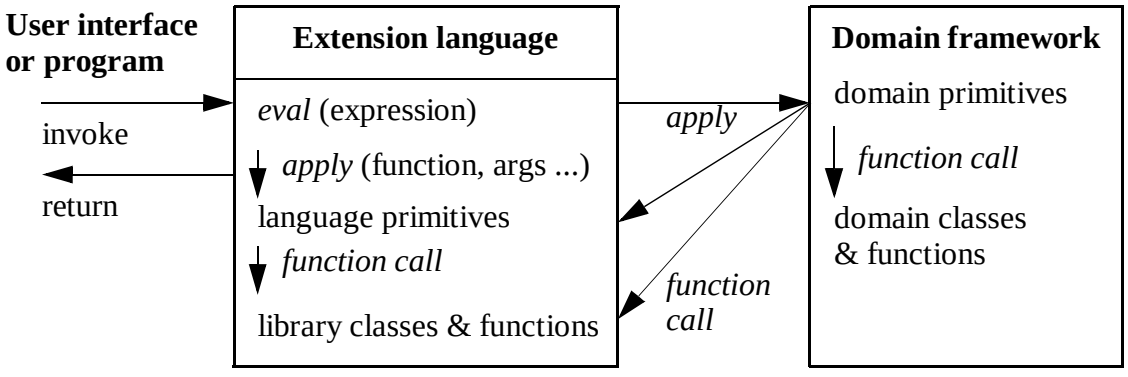

Figure 1: Calling patterns in a domain framework - extension language system

function invocations. Tcl then interprets byte codes for built-in primitives, and it forwards commands that start with registered primitive names to luxdbg. Like all Tcl functions, a luxdbg primitive returns a result string or error diagnostic upon completion of its invocation. Tcl can insert a result from a luxdbg primitive back into a higher-level expression; Tcl exception handling can catch luxdbg errors.

Figure 2 shows four classes of luxdbg primitives. *Processor management* primitives allow users to create, locate, initialize and destroy multiple processor instances. Processor instances include simulation models or connections to hardware processors of assorted types. *Processor access* primitives allow users to read and write processor state found in processor memory, registers and signals. *Processor control* primitives allow users to set and clear breakpoint event triggers, to handle breakpoint and error exceptions, to reset a processor, and to resume processor execution. *Processor IO* primitives allow users to connect input or output from processor input-output ports to data files or Tcl callback functions.

Exception processing and input-output processing provide two interesting examples of *event-driven callbacks* from the luxdbg domain framework to the Tcl extension language. A user can enable callbacks by associating a Tcl expression string with a breakpoint, with a processor error, or with a processor IO event. When one of these events occurs, luxdbg calls back to Tcl, passing a processor-event-expression triplet. Tcl evaluates the expression in the context of the processor and event. Callbacks can extend processor capabilities. An output callback, for example, can copy results from an output port to the input port of another processor, simulating interconnection. A breakpoint callback can log debugging information. Callbacks can access multiple processors. All callbacks can conditionally continue or halt processor execution. Tcl callbacks can consequently implement a multiprocessor simulation scheduler.

These four categories of primitives combine with domain event-driven control to transform Tcl from a vanilla programming language to a domain-specific processor manipulation language. Tcl expressions evaluated within callback functions can extend or override the native computations of processors.

3.2 Tcl-Luxdbg limitations

Luxdbg as diagrammed in Figure 2 has a number of limitations.

3.2.1 Ad hoc primitive interface code

Connecting a new luxdbg primitive to Tcl requires writing new, special-case code. Tcl forwards commands to luxdbg through the following interface function:

```
int Primitive(ClientData clientData, Tcl_Interp *interp,
    int argc, char *argv[])
```

The *clientData* parameter is a C void pointer that the domain framework initializes when it registers a primitive with Tcl. Thereafter Tcl supplies this pointer as clientData when it calls the primitive. The pointer is useful for passing domain state information.

The *interp* parameter is a reference to the Tcl interpreter. It is useful for eval callbacks and calls to Tcl primitives and library functions.

The remaining two parameters, *argc* (argument count) and *argv* (argument vector), are standard fare for C programmers. Argument vector is a vector of strings and

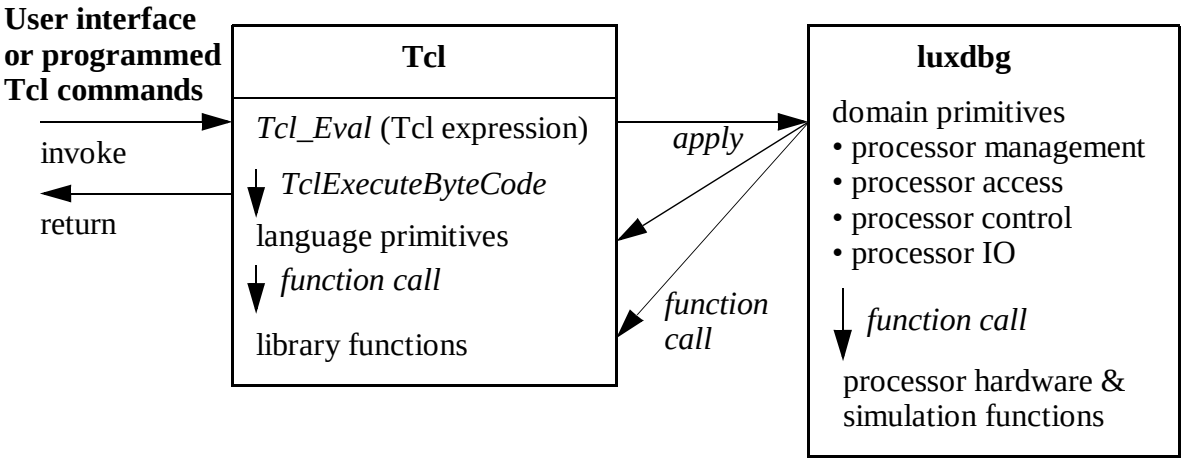

Figure 2: Calling patterns in the Luxdbg framework - Tcl language system

argument count holds its length. In calling a domain-specific primitive, Tcl stores the name and arguments of the primitive in argv. The primitive receives the arguments as strings, and the primitive has the responsibility of converting the strings to their appropriate values. Values might include strings as received, atomic types (e.g., int or float), Tcl-specific structured strings (e.g., Tcl lists), or domain-specific structured strings (e.g., an infix expression string for a processor debug statement). When a conversion error such as an invalid integer string occurs, the primitive has the responsibility of detecting the error and asserting a Tcl exception.

Tcl version 8.0 introduced an additional primitive interface for simplifying conversion of argument strings to atomic C types and for type mismatch detection [2]:

```
int TclPrim(ClientData clientData, Tcl_Interp *interp,
    int objc,Tcl_Obj *CONST objv[])
```

This interface replaces the string-based argv parameter with an array of *Tcl objects*. Client code can request conversion of one form of an object to another (e.g., a numeric string to an integer), but responsibility for directing argument type conversion and raising format exceptions still lies with the procedural code of primitives.

The need for testing parameter type-specific formats, directing translation of argument strings into objects of these types, verifying the correct number of arguments, and formatting return values, has resulted in a layer of luxdbg code to satisfy this need. Each new primitive requires a function to test and to convert its Tcl arguments to argument types required by its corresponding implementation function. The measurable cost is 1537 lines of non-comment code for 48 primitive functions in the current luxdbg, or about 32 lines of code on average for each primitive. While 1537 lines is a small number when compared with the roughly 80,000 lines of processor-neutral simulator and debugger code currently in luxdbg, the need to perform Tcl argument manipulation within each primitive contributes complexity out of proportion to the number of lines of code. Each designer of a primitive is saddled with the job of writing argument conversion code that has nothing to do with the primitive's semantics. Because this code comes in little, unrelated packets, each of which is specific to its primitive, it is written in an ad hoc manner. The process invites errors and time-consuming debugging and correction.

In addition, the creation of other tools that use Tcl as their extension language in our organization is causing the amount of code involved in ad hoc argc-argv manipulation to grow. Section 4 shows that Java reflection and a simple naming convention for primitives can eliminate this code.

3.2.2 Hard-coded dependence on Tcl

Several research-oriented luxdbg users have expressed interest in using the Scheme and Python languages with luxdbg. Nothing about the architecture of Figure 2 precludes replacing Tcl with a different extension language, but there are some hurdles. ELK-Scheme [1] and Python [3] use the following primitive interfaces:

```
Object ELK_Constarg_Primitive(Object firstArg,
    Object secondArg /* , etc. */)

Object ELK_Vararg_Primitive(int argc, Object *argv)

Object ELK_Lazyeval_Primitive(Object arglist)

int PyArg_ParseTuple(PyObject *argv, char *format,...)
```

ELK provides the first three, distinct interfaces. The first interface organizes arguments to a primitive call precisely by number of parameters supplied as part of primitive registration. The second interface allows a variable number of arguments. Both types use *eager evaluation*, resolving arguments to built-in Scheme types before calling the primitive. This is standard call-by-value evaluation. ELK passes arguments as dynamically-typed Scheme objects. The second interface supports variable-length argument lists and optional parameters. The third ELK interface uses *lazy evaluation* to pass a list of unevaluated Scheme objects to the primitive. This is a call-by-name mechanism that relies on the primitive to resolve argument text to argument values.

Python's argv is a *Python tuple* that contains call-by-value arguments. *Format* is a type conversion string similar to C's scanf function's format string [10]; it uses type conversion characters to direct type conversion of Python arguments into C atomic values and strings. Format implements a run-time interface definition language (IDL). Arguments following format are addresses of C variables that receive the results of format-directed conversions; there is no compiler or run-time check to ensure that format strings match the types of these variables. Python supports optional arguments and variable-length arguments, but PyArg_ParseTuple's format is limited to a fixed maximum number of arguments.

This variety in extension language-to-primitive interfaces presents a problem for designing a multiple extension language interface for a domain framework. A unique domain interface might be required for each

extension language. There is an underlying similarity among these interfaces, however, that is of assistance. Each invokes a primitive function with an argument list of *extension language objects*. An object may be a *Tcl string*, a *Tcl_Obj*, an ELK *Object* or a Python *PyObject*, but it is some type of an extension language object. Multiple extension languages result in a two-dimensional type system, where the set of extension languages defines one dimension (set of types $type_x$ = {Tcl, ELK, Python, ...}) and the union of all extension language internal types defines the other (set of types $type_y$ = {integer, float, string, sequence, ...}). An object's type is an element of $type_x \times type_y$.

A simple solution would be to design yet another primitive functional interface, this one for the domain framework. Each extension language would require a language-primitive-to-domain converter ($type_x \times type_y \rightarrow type_{domain}$), that would map its proprietary object format into the domain object format, thereby eliminating the $type_x$ dimension. One problem with this approach is the need to design $type_{domain}$. Another problem is the overhead of constructing an intermediate copy of each object in this language-neutral format. Section 4 shows that Java incremental class loading and reflection can support automated ($type_x \times type_y \rightarrow type_{Jy}$) conversion, where $type_{Jy}$ represents Java classes $\cup$ Java primitive types, and can include domain-specific classes. No intermediate format is necessary.

3.2.3 Unnecessary interpretation overhead

Every luxdbg command goes through extension language interpretation, resulting in execution that is slower than necessary.

In a scenario where a user enters commands directly into a terminal or commands come from an extension language script, interpretation overhead is necessary and acceptable. A user or program can supply any valid combination of commands, literal strings, control constructs (e.g., "if" statements), variable names and nested procedure calls. The extension language interpreter must resolve control flow, variable substitutions and return values from nested calls before calling domain primitive functions.

In luxdbg, however, typical interactive usage consists of a user interacting with a graphical user interface (GUI). The GUI forwards command strings to Tcl and receives result strings in reply. Tcl relays display update events from luxdbg to the GUI. Interpretation overhead is necessary in some cases, but for many cases it is not. Many of the commands coming from the GUI are *stereotyped commands*. These commands do not entail extension language control constructs, variables or procedure calls. One example is a button for resuming processor execution; it always sends the "resume" command, which Tcl interprets and sends to a luxdbg primitive. Another example is a text box for modifying a processor register value; it always sends a register assignment command with a value entered by the user, and again Tcl interprets the command and sends it to luxdbg without changes. Most commands attached to GUI objects are stereotyped commands that Tcl passes to luxdbg unchanged. These could go directly from the GUI to luxdbg, avoiding Tcl overhead.

Unfortunately, the primitive function interface of luxdbg in Figure 2 is not uniform. Direct GUI-to-primitive function calls would entail detailed encoding, within the GUI, of parameter type signatures of all luxdbg primitive functions. Ad hoc code for connecting specific GUI buttons, menus and text boxes to specific primitive functions and parameters would proliferate. Tcl strings provide a uniform command medium that avoids this proliferation, albeit at the cost of interpretation overhead.

Once again the simple solution proposed for dealing with multiple extension languages suggests itself. A GUI could encode a stereotyped command as a function name and list of arguments using the common object format $type_{domain}$, and the domain framework would complete the job by mapping $type_{domain}$ types to primitive parameter types. Section 4 shows an extension language-neutral way to reuse any extension language's internal types as $type_{domain}$, thereby avoiding the creation of a framework-specific $type_{domain}$.

4. Java reflection supports an extension language-to-domain bridge

The limitations in the current implementation of luxdbg as well as opportunities afforded by Java infrastructure have set me on the path of replacing the upper, debugger and profiler layers of luxdbg with a new design in Java. Lower processor modeling and hardware interface layers remain in C++. Anticipated Java-enabled improvements include the following:

- Better networking support for distributed debugging.
- Dynamic configuration through incremental loading.
- Reflection-based extension language interface support.

This section looks at ways in which Java's incremental loading and reflection can help to overcome luxdbg's existing limitations.

4.1 Java reflection and a naming convention eliminate ad hoc primitive interface code

4.1.1 Java reflection

Reflection is a mechanism whereby parts of a software system can query the system itself, in addition to the usual ability to query for application domain information. Whereas domain queries are the typical queries of any query-supporting system, reflective queries are meta-queries [11]. Meta-queries ask the question: "How does this system do some particular thing?"

Reflection provides support for self-configuring tools and utilities. A generic tool or utility can read reflection information and adapt itself to its target system. JavaBeans provide a popular example [6]. A Java class that conforms to certain method naming conventions, and that may provide additional compiled information that describes the class, makes itself available for manipulation by graphical design tools. JavaBean system developers can instantiate objects, set object properties and create inter-object communication paths using tools that contain no encoding of the APIs or semantics of particular JavaBeans classes. Instead of hard coding target class dependencies, JavaBean tools encode knowledge of the JavaBean reflection API. At system design time these tools read JavaBean class-specific information through the API, giving customers access to the unique capabilities of each JavaBean class in a set of beans.

Java reflection allows Java code to query about available Java classes, interfaces, methods, fields, and their properties at run time [12]. Figure 3 shows four major reflection classes — Class, Object, Method and Field — along with their associations and several methods that are important for this discussion. There are many more classes and methods in package java.lang.reflect.

Every Java object inherits from java.lang.Object. A reflection-based tool or utility calls Object.getClass to get a domain object's Class (java.lang.Class). With the object's Class in hand, a utility can determine constructors, superclass, implemented interfaces, nested classes, levels of protection, and most importantly for this discussion, methods and fields. Class.getMethods returns an array of Method objects, Class.getField returns a specific named Field, and Class.getComponentType returns the base type of an array. These are a few of the methods in class Class.

A Method supplies its name as a String, its parameter types as an array of Class objects, and its return type as a Class object. Clients of java.lang.reflect can build an array of correctly-typed Object arguments, and then call Method.invoke to invoke a Method on those arguments. Method.getParameterTypes provides the basis for automating the ($type_x \times type_y \rightarrow type_{Jy}$) conversion. Method.invoke provides the basis for automating domain primitive invocation.

A Field supplies its name as a String, its type as a Class object, and an assortment of get and set methods for retrieving and modifying its value. Reflective field query provides the basis for determining optional domain primitive parameters as part of automating the ($type_x \times type_y \rightarrow type_{Jy}$) conversion.

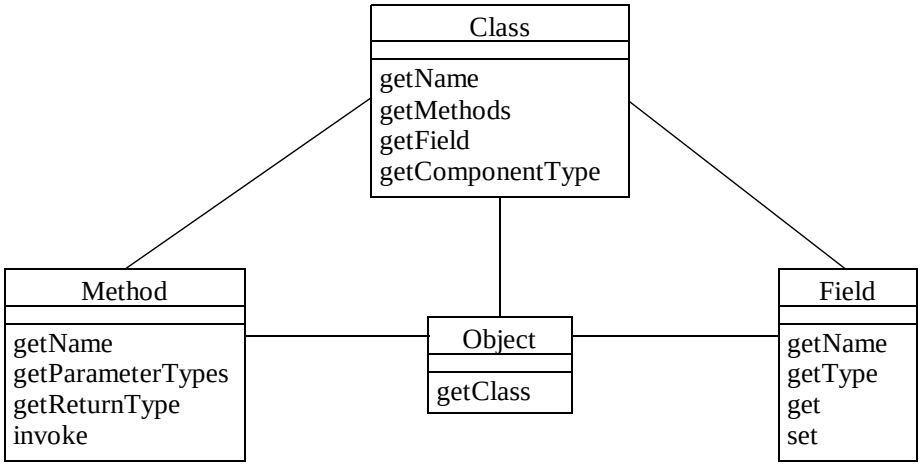

Figure 3: Central Java reflection classes

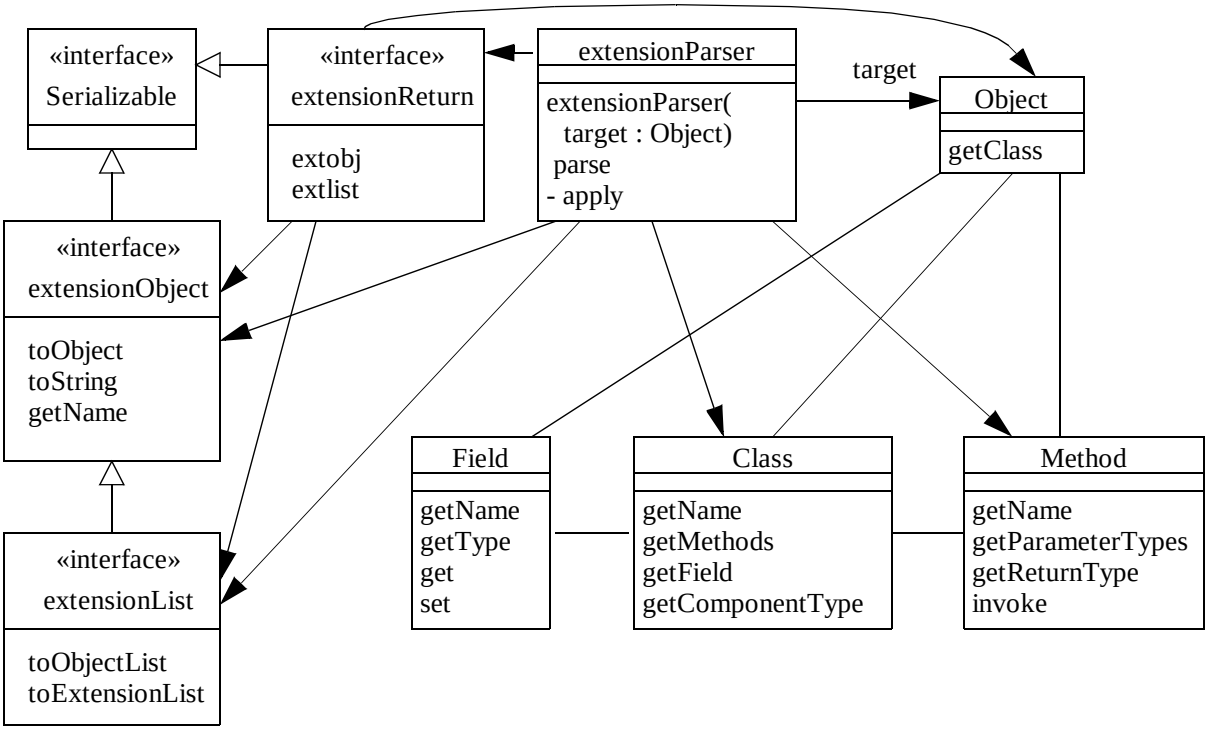

Figure 4: ExtensionParser translates extension language primitive calls

4.1.2 Eliminating ad hoc interface code

Figure 4 uses the Unified Modeling Language (UML) [13] to illustrate the major classes for ($type_x \times type_y \rightarrow type_{Jy}$) conversion and primitive method invocation. Hollow arrows signify inheritance, pointing from derived classes or interfaces to their parent classes or interfaces. Solid arrows are associations annotated for navigability. Clients navigate to the classes that serve them.

Class extensionParser is the central, client class of Figure 4.[1] ExtensionParser has three main methods: its *constructor*, its *parse* method, and its private *apply* method ("-" signifies private). The constructor takes a *target* Object as its parameter. The target is a domain object of any Java class type. Target makes domain primitives available to extensionParser. Each of target's primitive methods adheres to the naming convention "command_NAME," where NAME is the command name from an extension language's perspective. For example in luxdbg,

String command_stepi(int stepcount)

is a target method that implements the "stepi" command for stepping a processor "stepcount" machine cycles, returning processor status as a String upon completion.

At construction time extensionParser uses target.getClass to get the domain object's class, then it uses Class.getMethods to search the class for "command_" prefixed public methods. The constructor stores these in a hash table that is keyed on method name (without the "command_" prefix). Java permits method name overloading, and a slot in the hash table can hold multiple method references when that slot's method name is overloaded.

Command parsing relies on the interfaces on the left side of Figure 4. A Java interface defines public method signatures, but it has no body. Another interface can extend an interface's set of methods, and a class can implement the methods of any number of interfaces. In Figure 4 interface *extensionObject* represents an *extension language object* such as a *Tcl_Obj*, an ELK *Object* or a Python *PyObject*. Interface *extensionList* represents an ordered sequence of extensionObjects. Interface *extensionReturn* is a utility interface for converting Java objects into extensionObjects and extensionLists. ExtensionReturn is an inverse mapper

1. A working example of the code for extensionParser will be available as part of the on-line proceedings of this conference.

that formats return values when returning from a Java primitive to an extension language caller. The fact that these interfaces extend interface java.io.Serializable means that extensionObjects, extensionLists and extensionReturns can be passed as value parameters in networked method calls and stored as persistent objects.

Method extensionParser.parse receives an array of extensionObjects as an input parameter from an extension language primitive call. The first array element holds the command name, and the remaining elements hold its arguments. ExtensionParser.parse calls extensionObject.getName — all extensionObjects can be converted to Strings — and matches the returned command name to the method names stored in extensionParser's hash table. A hash slot gives a list of candidate methods that match the command name. Parse also receives an extensionReturn object from the extension language for formatting parse's return value.

The methods of extensionObject and extensionList come into play when parse calls extensionParser.apply. Apply is a recursive, backtracking match algorithm inspired by the more powerful match algorithms of PROLOG [14] and ML [15]. Apply takes as arguments the *command name*, a matching *candidate Method*, an input array of *extensionObject primitive-arguments*, another input array of *Method parameter types* (obtained in parse via Method.getParameterTypes), an array of *domain Object arguments* that apply populates by translating the extensionObject array, and parse's *extensionReturn parameter*.

Each recursive call to apply attempts to match the next extensionObject argument to the next Method parameter type. Apply relies on an extension language-specific class that implements interface extensionObject to do the hard part. ExtensionObject.toObject takes the Method parameter type as its argument, and it attempts to convert itself into a Java Object of that type, returning that Object as its return value. Conversion starts by matching basic extension language object types to basic Java Method parameter types such as integers, booleans, floats and strings. When simple conversion fails, apply uses reflection to determine whether the Method parameter type has a class-static *valueOf* method that can convert an extensionObject into a domain Object. Target domain classes can provide custom extensionObject-to-domain Object converters by defining valueOf. ExtensionObjects transform into specialized domain-class objects without encoding domain awareness into extensionObject classes. Finally, if toObject cannot transform its extensionObject into the required domain Object, toObject throws a *typeMismatch* exception to apply.

A concrete extensionObject.toObject method is doing most of the work of ($type_x \times type_y \rightarrow type_{Jy}$) conversion. There is one such concrete method for each extension language in $type_x$, and by operating behind the abstract extensionObject interface it eliminates its specific $type_x$ language from extensionParser's view. A concrete extensionObject.toObject method must be written once for each extension language. Its availability simplifies ($type_x \times type_y \rightarrow type_{Jy}$) conversion to ($type_y \rightarrow type_{Jy}$) conversion at the time that parameter matching occurs. The extension language internal type of $type_y$ maps itself to the domain type of $type_{Jy}$; the latter is extensionObject.toObject's Method parameter type argument.

ExtensionParser.apply uses some mechanisms in addition to extensionObject.toObject. Apply checks whether the target Method accepts an extensionObject at the current position, passing an unaltered argument when acceptable. In this case the primitive must initiate conversion of the extensionObject at a later time. When a position match succeeds, apply advances to the next extensionObject and Method parameter type. When a position match fails, apply inspects a list of optional parameter positions built by extensionParser's constructor. The constructor uses the target Object's Class.getField method to locate an integer array "optional_NAME" for command method "NAME." If the array exists, each of its elements gives the offset of a parameter position that is optional for that command. Apply supplies a Java null reference for an optional position that cannot match the current extensionObject, then it continues searching.

If apply encounters an array Method parameter, it uses Java's *instanceof* predicate to determine whether its current extensionObject is in fact an extensionList, and apply uses reflection to determine whether the component type of the extensionList matches the component type of the parameter (obtained from Class.getComponentType). On a match, apply builds an array argument.

Finally, if apply arrives at the last Method parameter with a sequence of unmatched extensionObjects, and if the last Method parameter is an array, apply attempts to populate the array from these residual extensionObjects. A final array of type extensionObject receives the trailing arguments unaltered; apply invokes extensionObject.toString for a final array of type String.

If, at last, extensionParser.apply matches all parameters and consumes all extensionObject arguments, it uses Method.invoke to invoke the primitive method on its Java Object arguments. On success, apply uses extensionReturn.extobj to return the primitive result to

the extension language. When an applied method fails, apply and parse throw an exception back to the extension language.

Match failure, on the other hand, does not throw back to the extension language. Upon match failure, apply backtracks and attempts to use nulls for optional parameters; then it again works forward. If exhaustive attempts to match a particular Method fail, and if that Method name is overloaded, parse supplies another Method to apply. Only when all possibilities have been examined does parse report a usage error to the extension language.

Consider the example of Figure 5. This example shows two variants of a debugger *stop* command. Command "stop at location ?expression?" sets a breakpoint at a numeric processor address. Command "stop in function ?expression?" sets a breakpoint within a named function. Both primitives specify an optional callback expression to evaluate when the breakpoint fires. The callback is an interpreted extension language expression provided by a user.

This example is useful in pointing out what extensionParser both can and cannot do. It cannot distinguish a method on the basis of a textual keyword. Regardless of whether stop's first argument is "at" or "in," matching will pair it with the "String keyword" parameter of either method and attempt to use it. A more complex extensionParser mechanism would allow keyword-method pairs to be listed in a target Object field, and it would invoke a method only if its specified keywords are matched. It turns out that it is just as easy to have a matched method check its keywords on invocation, throwing exception *typeMismatch* on a keyword mismatch. ExtensionParser.apply treats typeMismatch as a mismatched method, and it continues searching for another match. This approach is similar to failure in the body of a PROLOG clause whose head has matched its arguments [14].

In the first attempt for method "stop," apply fails to match string "myfunc" to an int *location* parameter. Backtracking determines that the first parameter is not optional, and so apply backtracks to parse, which calls apply with another "stop" method. In this attempt, apply matches "myfunc" to the *function* parameter and the callback string to the *expr* parameter. Apply invokes the second command_stop method. If the callback expression had been missing, apply would have backtracked and inspected the optional parameter entry derived from optional_stop_3, filling *expr* with a null reference.

Note that while the callback expression {puts "stopped in myfunc" ; resume} is a Tcl list, nothing in the match algorithm encodes dependence on Tcl. The third extensionObject argument happens to be an extensionList that happens to be a Tcl list, but the concrete realization of this extensionObject as a Tcl list is unknown to extensionParser. The syntax handling for Tcl list construction occurs in the Tcl interpreter before the primitive call begins. Domain object access to list elements can occur through extensionList.toObjectList or extensionList.toExtensionList, converters that strip off language-specific list syntax and return an array of domain objects or an array of extension language objects respectively. In this example the domain object simply stores the Tcl list without decoding its values. When a domain processor reaches a breakpoint it passes the callback to the extension language (without encoding its identity as a Tcl interpreter) for evaluation.

This section has shown how Java reflection and a naming convention on method names eliminates hand coded parameter conversion code for primitive methods. ExtensionParser aligns parameters and reports errors. The next section shows how dynamic class loading combines with the reflective capabilities of

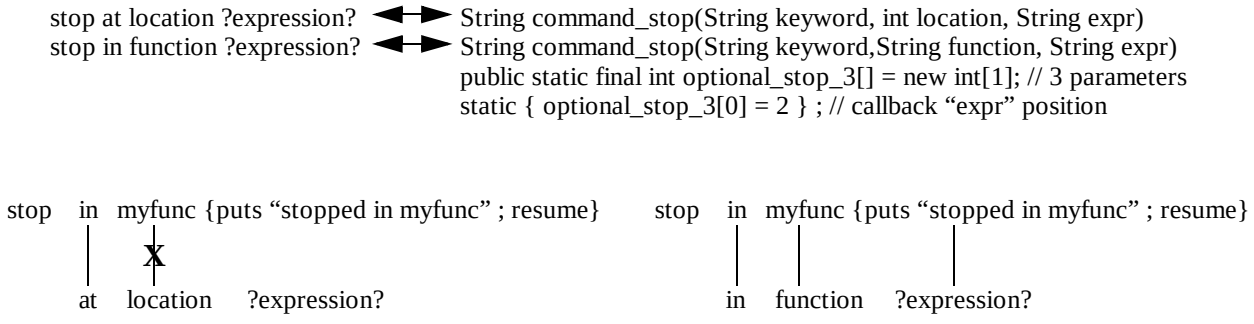

Figure 5: An example of automatic primitive method parameter alignment

extensionObject to support extension language selection at run time.

4.2 Extension language as a parameter

Figure 6 shows interfaces *extensionObject*, *extensionList* and *extensionReturn* of Figure 4, and it also shows interface *extensionLang* that encapsulates an extension language interpreter. ExtensionLang includes public methods for interpreting expressions and applying functions. Figure 6 also shows Tcl implementation classes that implement these four interfaces. The dashed lines signify UML *realization* equivalent to Java's *implements* directive. TclObject, TclList and TclReturn assist in converting Tcl objects to Java Objects and in returning Java Objects to Tcl as discussed in the last section. TclInterp houses a Tcl interpreter that uses TclObjects, TclLists and TclReturn to communicate with Java primitive methods.

The Tcl classes of Figure 6 support Tcl by wrapping the C implementation of Tcl 8.1.1 with Java Native Interface (JNI) proxy methods [16]. Each proxy method calls its Tcl counterpart through JNI's C binding. Tcl does not encode dependence on Java, and most Java classes that use extension languages encode dependence only on the abstract interfaces of Figure 6.

The four classes TclInterp, TclObject, TclList and TclReturn constitute a *Tcl software component*. Collectively they implement the four interfaces needed to install an extension language in this Java framework. Furthermore, Java's ClassLoader.loadClass method allows this Java framework to load a specific extension language, by name, at run time. The luxdbg extension language loader appends the string "Interp" to the extension language name (e.g., "TclInterp"), loads that specific language class from the luxdbg package that houses extension language components, and performs a run-time type check to ensure that the loaded class implements the extensionLang interface. ExtensionLang uses the other interfaces of Figure 6, and loading an extension language also loads the other concrete classes. Loading TclInterp loads TclObject, TclList and TclReturn classes as well. The loader is similar to reflection in supporting a string-based approach to determining available extension languages at run time. A GUI could inspect available languages in luxdbg's extension language package and allow a user to select

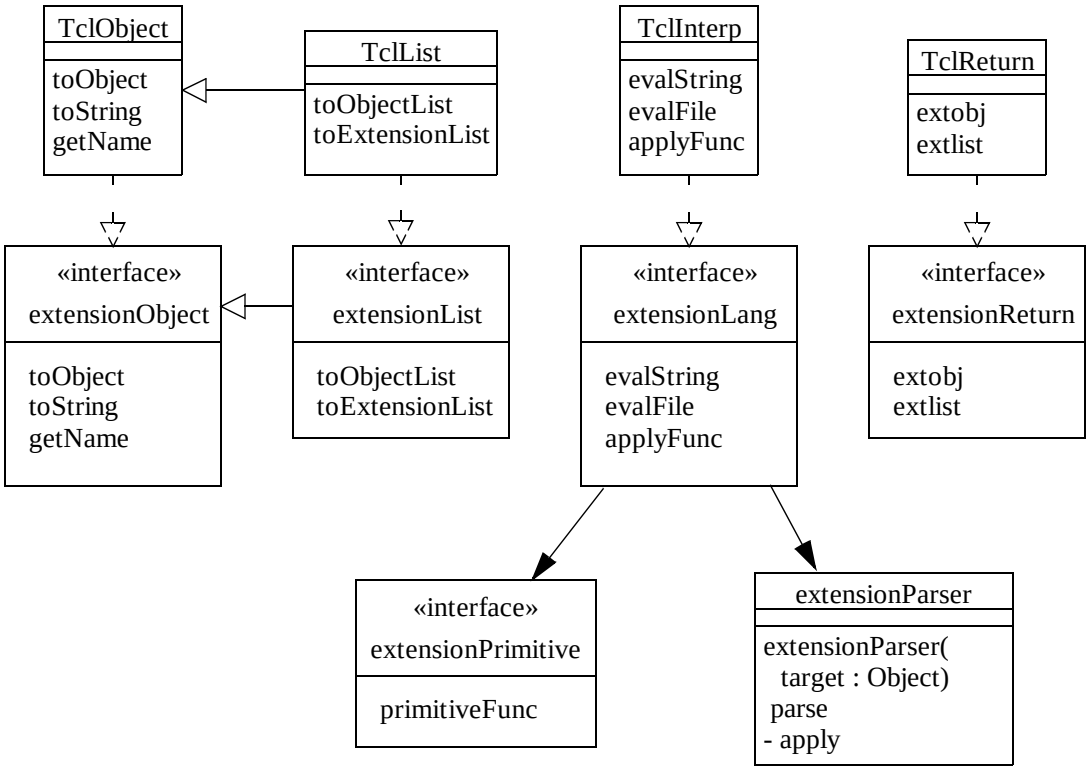

Figure 6: ExtensionLang encapsulates an extension language

the language of choice.

ExtensionLang defines helper interface *extensionPrimitive* that specifies method *primitiveFunc*. ExtensionPrimitive.primitiveFunc has the same signature as extensionParser.parse. For the TclInterp example, TclInterp queries its extensionParser for "command_" primitive names at construction time, and it registers each primitive command with its C-level Tcl interpreter. Registration includes a pointer to a JNI function that, when called as a primitive from Tcl, calls TclInterp's version of extensionPrimitive.primitiveFunc. Tcl delegates primitive commands to a C-level JNI function, which in turn delegates to TclInterp's extensionPrimitive.primitiveFunc, which in turn delegates to extensionParser, which then performs the matching and method invocation discussed in the last section. Return values and exceptions come back the delegation chain. TclReturn converts return objects to Tcl objects, and the JNI function converts Java exceptions to Tcl exceptions.

Thus, four straightforward interface abstractions — an extension language, its objects, object sequences, and return values — suffice to encapsulate a complex language as a Java software component. The $type_x$ term has become a run-time parameter that users can set.

4.3 Commands that bypass the interpreter

Section 3.2.3 raised the issue of unnecessary interpretation overhead. Stereotyped commands from a GUI need not go through an extension language interpreter because the extension language does not change stereotyped command strings. The alternative of connecting a GUI directly to domain object implementation methods is undesirable because it over-couples GUI code to primitive method signatures. GUI code becomes ad hoc. Section 3.2.3 proposed that a GUI could encode a stereotyped command as a function name and list of arguments using the common object format $type_{domain}$, and the domain framework would complete the job by mapping $type_{domain}$ types to primitive parameter types.

ExtensionParser.parse is precisely the method needed to provide a uniform command interface that bypasses the extension language interpreter. Parse's main input parameter is an array of extensionObjects to translate. A Java GUI can map user interface events (e.g., button pushes, etc.) → arrays of Strings, then map Strings → $type_{domain}$ objects in the form of extensionObjects by calling extensionReturn.extobj, then send an array of extensionObjects to extensionParser.parse; parse then maps $type_{domain}$ objects as extensionObjects → $type_{Jy}$ via extensionParser.apply and extensionObject.toObject. All extensionObject classes can hold strings (string representation is possible for all extension language types), so mapping Strings → $type_{domain}$ objects entails no type conversion overhead.

Going back to Figure 1, a User Interface can bypass the Extension Language component and send all stereotyped commands directly to the Domain Framework's extensionParser interface. Two design patterns from the Gang of Four book are conspicuous here [17]. ExtensionParser implements the *Facade Pattern*. ExtensionParser provides a unified, homogeneous interface to a set of heterogeneously typed primitive domain methods. Next, extensionParser.parse's input array of extensionObjects implements the *Command Pattern*. The first array element is a command name and the remaining elements are its arguments. Command arrays can be stored, queued, forwarded and ultimately executed via extensionParser.parse.

The only potential drawback to bypassing the extension language is the fact that users cannot extend or otherwise redefine stereotyped commands in the extension language if those commands always bypass the extension language. Luxdbg avoids this problem by registering primitives, including stereotyped command names, with the extension language, and then having the extension language notify UI components if any stereotyped commands are redefined. At that point those commands are no longer stereotyped. After redefinition, UI components must send these commands through the extension language component.

4.4 Performance

The current Java implementation of luxdbg transforms the C++ implementation of Figures 1 and 2 into associations of UI and Domain Framework Java components interconnected by extensionLang, extensionObject, extensionList, extensionReturn and their Tcl concrete counterparts. UI-Domain communications ultimately pass through extensionParser to luxdbg's Domain Framework. How much does all this encoding, message passing, and reflection-based decoding cost?

The answer is that, compared to extension language interpretation costs, reflection-based command parsing is cheap. Table 1 summarizes the results of sending 250,000 command calls through each of three interfaces:

- direct UI-to-Domain object method calls with no extension language or extensionParser involvement

- construction and passage of command objects (i.e., extensionObject arrays) from UI strings to extensionParser as discussed in Section 4.3, bypassing the extension language interpreter
- evaluation of UI command strings in the extension language interpreter, which uses extensionParser to decode primitive calls

The target primitive method takes a single integer parameter and it returns a constant Java String. The machine is a lightly loaded Toshiba Tecra laptop with a 266 MHz Pentium processor, 96 Mbyte RAM and 32Kbyte internal cache, running Windows 95, Sun's Java Development Kit 1.2 and Tcl 8.1.1. The test driver invokes Java's garbage collector immediately before each of the three measured 250,000-call series. Table 1 reports time-per-call in microseconds.

Table 1: μSeconds-per-call for direct calls, command objects and interpreted expressions

test	direct	parsed command objects	Tcl 8.1.1 interpreter
argv, 1 method	0	42	399
argv, 50 methods	0	29	378
Tcl_Obj, 1 method	0	43	417
Tcl_Obj, 50 methods	0	36	391

The first two rows use Tcl's original string-based, char **argv primitive interface. The last two rows use the newer Tcl_Obj object interface that attempts to keep objects in an appropriate primitive format (e.g., string, int or float) until the object is needed. The first and third rows define only 1 method, the test target method, in the test domain object. The second and fourth rows define 50 primitive methods, including 2 additional, overloaded instances of the target method name with different parameter types.

All rows show that 250,000 calls were not enough to bring direct call overhead out of the noise. The 0 figure does signify that communication and interpretation overhead accounts for all measurable delays in other columns.

Average extension language interpretation runs about 10.6 times slower than command objects that bypass the interpreter. Clearly overhead is eliminated. At 29 to 43 microseconds of command overhead per GUI event, command objects that bypass the extension language are clearly fast enough. There is no reason to over-couple the GUI to the Domain Framework for speed.

378 to 417 microseconds of interpreter overhead includes the call to extensionParser.parse on the Domain Framework side of the extension language. Roughly 400 microseconds per call is still not a lot of overhead. C++ luxdbg has the additional problem that the extension language extracts all Domain Framework-to-GUI update events and sends them to the GUI. Java luxdbg will eliminate the extension language from stereotyped GUI callbacks as well.

The first surprise comes with the fact that the second row, working with a more heavily populated extensionParser Method hash table than the first row, is nevertheless faster than the first row. This result was consistent across tests, and it is repeated between rows three and four. The only conclusion is that Java hash tables are marginally more efficient when populated with a typical command set size.

The next surprise comes with the fact that the first two rows, last column, are marginally more efficient than their counterparts in the last two rows. The char **argv implementation of Tcl objects in the first two rows translates Tcl objects to Java strings immediately upon leaving Tcl to invoke a primitive. The Tcl_Obj implementation of Tcl objects in the last two rows stores a reference to a C-level Tcl_Obj in each Java TclObject. It does not translate a Tcl_Obj into a domain Object until TclObject.toObject runs, and it converts it directly to the integer needed by the test method. It skips the intermediate format of a Java String. The benefit of avoiding the intermediate String format appears to be offset by the fact that TclObject.toObject must call through the Java Native Interface to C in order to extract the integer value by calling Tcl's Tcl_GetIntFromObj library function. JNI calls add overhead. The TclObject caches its value to avoid subsequent calls through JNI, but typical extensionObjects (e.g., TclObjects) require only one toObject call, so caching is not much help. Tcl's original char **argv is both simpler to program and faster for this application.

ExtensionParser cost is roughly one tenth the cost of extension language + extensionParser costs for calling a stereotyped command that does nothing.

ExtensionParser's percentage contribution to overhead diminishes as the extension interpreter is given real scripts to interpret, and as the Domain Framework is given real primitives to execute. Interpreter and domain costs go up while extensionParser costs remain constant. Clearly performance is adequate for extension language primitive interfaces and Command design pattern objects.

5. Related work and conclusions

5.1 Related work

Other existing Java-based implementations of Tcl include Jacl and TclBlend [18]. Jacl is a partial implementation of the Tcl interpreter in Java, while TclBlend is a conventional C Tcl implementation with an interface to Java. The report on Jacl and TclBlend states, "For the Java platform, we envision an architecture that includes Java as the 'component' language used by component developers and Tcl as the 'glue' language used by application assemblers." [18] That perspective appears to make Tcl the center of the framework. Tcl is responsible for interconnecting and synchronizing Java components. Jacl is purported to be slow [19] — Jacl interprets Tcl on top of a Java bytecode interpreter — although a Jacl implementation that produces Java bytecodes is certainly possible.

Luxdbg's use of Java and Tcl takes a different approach. Luxdbg uses Tcl where extension language interpretation makes sense, but it does not put extension language interpretation overhead in the middle of the architecture. Luxdbg's extension language interpreter has access to all primitives, and it is possible to coordinate all framework activities from Tcl, but it is not mandatory. JavaBeans construction environments do a reasonable job of generating "glue" code for stereotyped component interactions, saving the extension language for what it does best, extending the system at run time. Given Tcl's inability to coordinate multiple Java threads within a single Tcl interpreter [20], Tcl is not an ideal candidate for the center of any multi-threaded Java framework.

Part of the impetus for creating the extension language-neutral extensionLang interface was the desire to experiment with JPython [21], a Java implementation of Python, within luxdbg. Interesting features of JPython include dynamic compilation to Java bytecodes for performance, and the ability to extend existing Java classes in JPython. JPython is tightly integrated into Java, and implementing extensionLang and the related interfaces should be straightforward. This is an area for future investigation.

5.2 Conclusions

This paper started out as an overview of using extension language components with application domain frameworks. It looked at a particular framework, luxdbg, and its coupling to the Tcl extension language. Integration of Tcl into C++ luxdbg has been a great success, but it has suffered from a few limitations. Interface code from Tcl to C++ primitives is often ad hoc and annoying to program, tight coupling of the luxdbg framework to Tcl limits its ability to work with other extension languages, and putting the extension language in the center of all UI-to-Domain interactions adds unnecessary overhead.

Java reflection and dynamic loading have provided the basis for a set of mechanisms that overcome these limitations. Reflection and a naming convention allow class extensionParser and interface extensionObject to work together to eliminate ad hoc primitive interface code. The abstract extensionLang interface and dynamic, name-based class loading work together to make the specific extension language in a system a run-time parameter. Command pattern objects in the form of extensionObject arrays support stereotyped GUI-to-Domain interactions that are uniform and efficient. Clearly Java reflection and dynamic loading are very powerful tools for enhancing the utility of extension languages.

6. References

1. "The Extension Language Kit (ELK)", http://www-rn.informatik.uni-bremen.de/software/elk/. ELK is an implementation of Scheme organized for use as an extension language.
2. Brent Welch, *Practical Programming in Tcl and Tk*, Second Edition. Upper Saddle River, NJ: Prentice Hall PTR, 1997.
3. Guido van Rossum, *Extending and embedding the Python interpreter*. Amsterdam: Stichting Mathematisch Centrum, 1995, also at http://www.python.org/doc/ext/ext.html.
4. John Ousterhout, "Scripting: Higher Level Programming for the 21st Century," *IEEE Computer*, March, 1998, or Scriptics Corporation, http://www.scriptics.com/people/john.ousterhout/scripting.html.
5. John Allen, *Anatomy of LISP*. New York: McGraw-Hill, 1978.
6. Robert Englander, *Developing Java Beans*. Sebastopol, CA: O'Reilly, 1997.
7. *LUxWORKS Debugger User Guide*, luxdbg Version 1.7.0, Lucent Technologies, December, 1998.

8. D. Parson, P. Beatty and B. Schlieder, “A Tcl-based Self-configuring Embedded System Debugger.” Berkeley, CA: USENIX, *The Fifth Annual Tcl/Tk Workshop ‘97 Proceedings*, Boston, MA, July 14-17, 1997, p. 131-138.

9. D. Parson, P. Beatty, J. Glossner and B. Schlieder, “A Framework for Simulating Heterogeneous Virtual Processors.” Los Alamitos, CA: IEEE Computer Society, *Proceedings of the 32nd Annual Simulation Symposium*, IEEE Computer Society / Society for Computer Simulation International, San Diego, CA, April, 1999, p. 58-67.

10. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Second Edition. Englewood Cliffs, NJ: Prentice Hall, 1988.

11. Frank Buschmann, “Reflection,” in *Pattern Languages of Program Design 2*, ed. J. Vlissides, J. Coplien and N. Kerth, Reading, MA: Addison-Wesley, 1996, p. 271-294.

12. Ken Arnold and James Gosling, *The Java™ Programming Language*, Second Edition. Reading, MA: Addison-Wesley, 1998.

13. James Rumbaugh, Ivar Jacobson and Grady Booch, *The Unified Modeling Language Reference Manual*, Reading, MA: Addison-Wesley, 1999.

14. W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*, Second Edition. Berlin: Springer-Verlag, 1984.

15. Jeffrey D. Ullman, *Elements of ML Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1994.

16. Rob Gordon, *Essential JNI: Java Native Interface*. Upper Saddle River, NJ: Prentice Hall, 1998.

17. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

18. Ray Johnson, “Tcl and Java Integration,” Sun Microsystems Laboratories, February 3, 1998. See http://www.scriptics.com/java/

19. Assorted discussions on comp.lang.tcl.

20. Tcl 8.1.1 documentation at http://www.scriptics.com

21 JPython home page at www.jpython.org